by
Jeff Stefan

Serial Communications and Protocol Stacks:

Part 1 — *Why All The Layers?*

This article explains the concept of layered protocol stacks and provides you with interrupt driven serial communications routines that you can use in your own applications.

Part 2 will provide you with a Data Link Layer protocol built from the low-level functions supplied here. The code is written using Turbo C, and supports transfer speeds up to 19.2K.

Serial programming is complex and requires knowledge of interrupt level programming and your computer's hardware. By using the code, examples, and concepts provided in this article, you will understand the PC's serial port hardware and be well on your way to inventing your own communications protocols.

Protocol Stacks

Reliable communications protocols are complicated and can be difficult to program. In early networks, most of the design effort went into hardware, and software was thought of as the easy or trivial part. Nowadays, hardware is (unfairly?) considered the easy element and software is definitely the complicated part.

In the early days of computing, a lot of smart people spent untold hours of frustration trying to get computers to communicate reliably. Along with the frustration came potentially vast rewards. By getting computers to effectively communicate with one another, programmers could draw on the unique resources and capabilities of the connected machines, creating a powerful virtual machine.

How was all this complexity to be managed in order to harness the power of a virtual machine? Divide the complexity into clear cut layers of functionality, starting from the most generalized abstraction at the top layer and becoming more and more specific as the bottom layer is approached.

That's the common sense idea behind dividing up computer communications into several layers. A conceptual model of a layered protocol stack is shown in Figure 1.

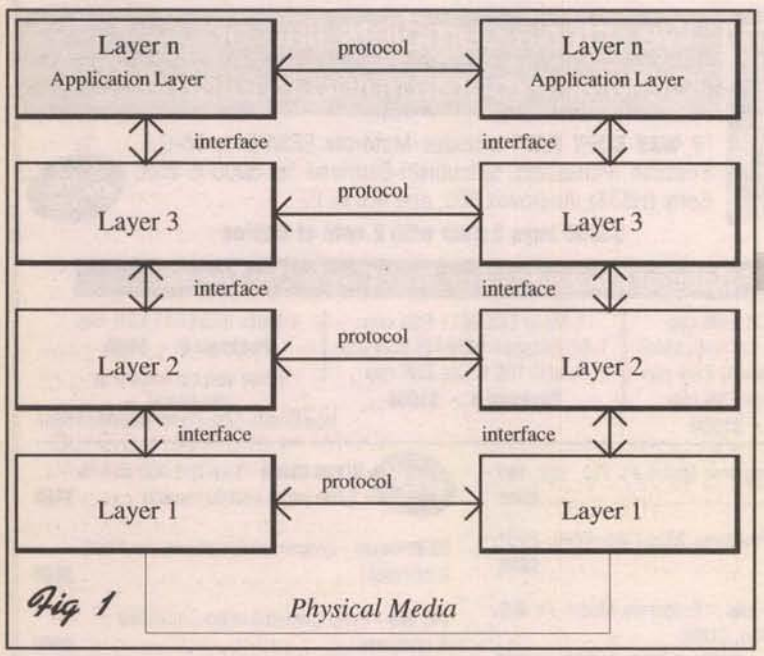


Fig 1

The key to understanding layered models is understanding the difference between a protocol and interface. A protocol is an agreed method of communication between layers. An interface is defined as the services available from one layer to the next.

Each layer in the protocol stack speaks exactly the same language, and can only communicate at the same layer level. This is sometimes called peer-to-peer communication, which is represented by the horizontal lines between the layers in Figure X.

Layer 1 on one machine is exactly like Layer 1 on another machine, Layer 2 is the same as Layer 2, and so on. The services increase in abstraction from the bottom up, all the way up to the top layer. The top layer is usually called the Application Layer. Each Layer, except for the Application Layer, offers clearly defined services to the other layers. The Application Layer only uses services, since there are no more layers above it. The services offered to each of the layers are made available via Service Access Points, or SAPs.

For example, let's say that within a program you call a function named ReadFile(filename) in the Application Layer on Machine 1. Figure 2 illustrates a simple protocol stack used to implement ReadFile(filename).

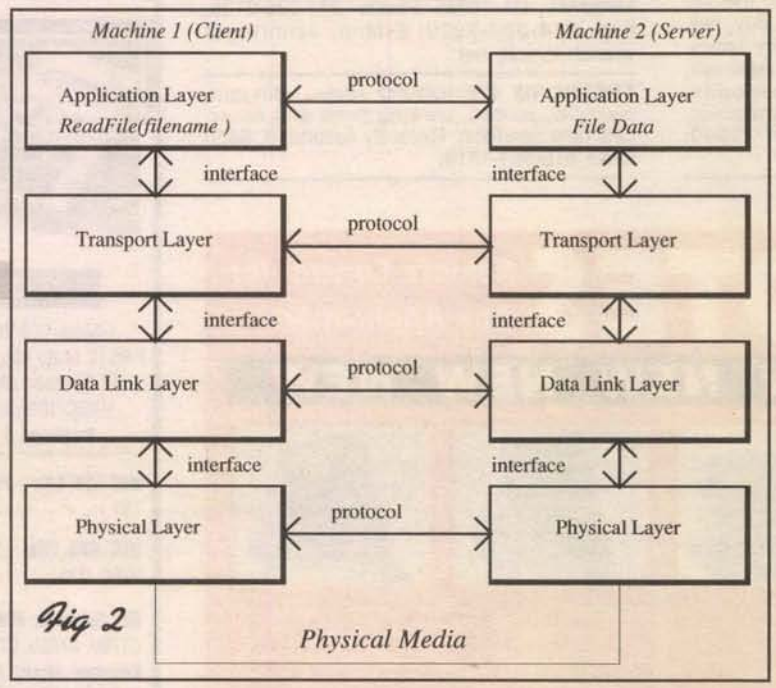


Fig 2

Assume that the actual file is on Machine 2. The Application Layer does not know, and does not care that the file is physically located on another machine. The Application Layer expects file data to be returned when it calls ReadFile(filename), nothing more, nothing less. It doesn't care about data packets, error checking methods, packet retries, or even the physical transmission media. That's the job of the lower layers.

The Application Layer is the most abstract, with each underlying layer becoming less and less abstract and more concerned with the actual work of transmitting and receiving the file. This is the idea behind Client/Server applications. The Application Layer on Machine 1 is the Client. It just wants file information, and doesn't care where it gets it from. The Server responds to the ReadFile(filename) function call and provides the file data.

In our model, we have four layers. The Application Layer is the highest level and does not provide any services to other layers. Next is the Transport Layer. The Transport Layer's responsibility is to provide reliable services to the Application Layer. It will take the file data, which is usually very large, and break it into smaller numbered segments to prepare it for transmission, as shown in Figure 3.

The Transport Layers on both machines understand that the file data will be received in enumerated segments, then will re-assemble the data segments back into a continuous file, in the proper order. The Transport Layer must ensure that the data is assembled in the right order.

Some of you familiar with the OSI and TCP/IP stacks may wonder why there is no Network Layer present. This article focuses on single machine-to-machine communications. There is no network to deal with.

The next layer in our model is the Data Link Layer. The Data Link Layer takes the raw input data stream from the Physical Layer if receiving, or takes fragmented data from the Transport Layer if sending, and breaks it up into frames. Data Link Layer frames typically are partitioned into two or three parts. The leading part of the frame is the header, the middle is the payload data, and the trailer is the frame's checksum or CRC (Cyclic Redundancy Check). The header contains different fields such as the frame's sequence number, the length of the frame, and an

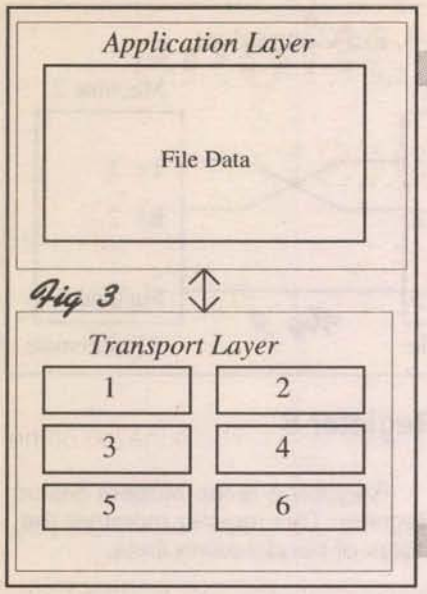


Fig 3

acknowledgement byte. The frame's integrity is checked using checksums, parity calculations, or CRCs. These values are usually one or two bytes and are placed in the trailer. A generic frame is shown in Figure 4.

(The Data Link Layer frames developed in Part 2 of this article will be based on these three segments.)

Going back to the protocol stack, the Transport Layer data chunks are passed to the Data Link Layer. The Data Link Layer encapsulates the data within a header and trailer, as shown in Figure 5. The data is now ready to be passed to the Physical Layer and sent across the transmission media.

As you can see, the functionality of each layer becomes more specific and detailed as it approaches the Physical Layer. The Transport Layer simply breaks the file data from the Application Layer into smaller segments. It has no knowledge about headers and trailers, but it knows about the size and order of the data segments. The Data Link Layer has no knowledge of the specific Physical Layer transmission media.

The transmission media could

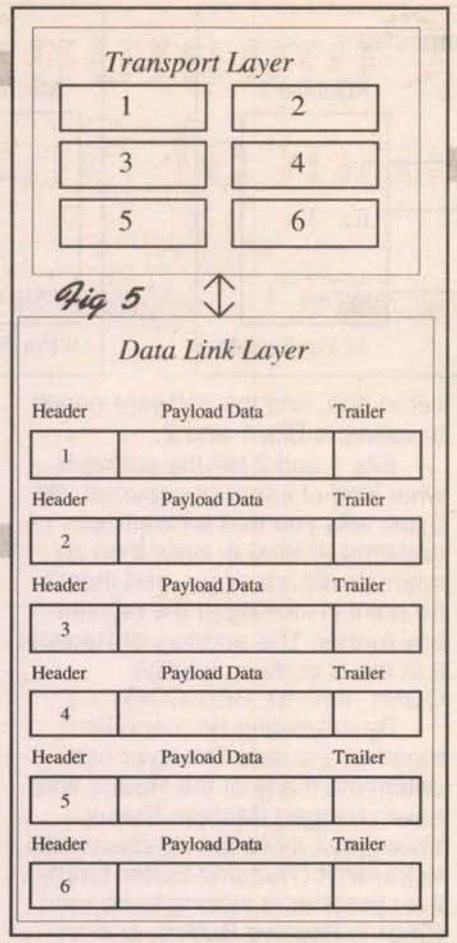


Fig 5

be a single wire, fiber optics, a parallel cable, or an RS232 port. The physical media may change, but that doesn't affect the Transport or Application Layers.

When the Data Link Layer protocol designers decide to implement a new hardware interface, they write and install new Data Link Layer software. This usually consists of setting hardware register values. None of the other layers care, or are even aware of the Data Link Layer change. Layered protocols were invented for transparency and interchangeability.

The Data Link Layer in our model is designed for RS232 communications and is divided into two sub-layers. These layers are the Input Queue sub-layer and the Message Buffer sub-layer, as illus-

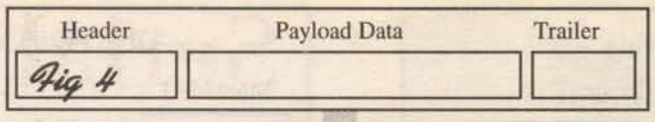


Fig 4

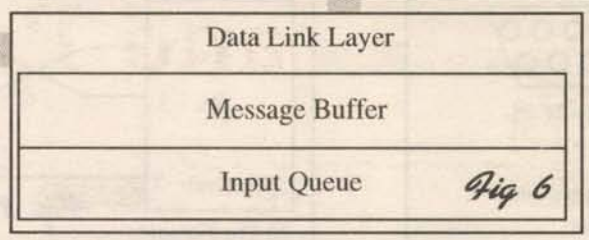


Fig 6

trated in Figure 6.

The Input Queue sub-layer is on the bottom and interfaces with the Physical Layer. Characters arrive from the Physical Layer and are placed in the Input Queue, which is a circular buffer. The Input Queue is declared as an array of unsigned chars, and is named InputQueue[MAX_QUEUE_SIZE].

MAX_QUEUE_SIZE is set to 1024 characters. Four pointers control characters entering and exiting the queue. The pointers are *HeadPtr, *TailPtr, *StartOfQueue, and *EndOfQueue.

The *HeadPtr and *TailPtr act as the service access points between the Input Queue level and Message Queue levels of the Data Link Layer.

The *TailPtr places the characters received from the UART receive register in the queue, and the *HeadPtr extracts characters from the queue. The *TailPtr resides in the interrupt service routine.

When the *TailPtr reaches the end of the queue, it wraps around to the start of the queue. The *HeadPtr removes the character from the Input Queue and puts it in the Message Buffer.

When the *HeadPtr reaches the end of the queue, it also is reset to the start of the queue. That's why the pointers *StartOfQueue and *EndOfQueue are needed. They hold the addresses of the start and end of the Input Queue.

Functions within the Message Buffer sub-layer detect the mes-

sage frames, extract the header information, and perform error checking. These functions, along with simple data packet transfer protocol functions, will appear in Part 2.

The RS232 Physical Layer

The most common way of getting two computers to communicate is using the available RS232 COM ports. Older PCs and a lot of equipment use a 25-pin connector, where most newer PCs use a 9-pin connector. A DB-25 (or 9-pin equivalent) male connector is mounted on the PC side. The pinouts for each connector are listed in Figure 7.

The code for this article doesn't use any software or hardware flow control, so the cable we'll use is a simple three-wire connection. Our cable consists of Tx (Transmit Data), Rx (Receive Data), and Signal Ground. The cable pinout for the 25- and 9-pin connectors are illustrated in Figures 8 and 9. These are the cables that go between machines, so they require female connectors.

If you don't have two machines to play with, you can make a simple loopback plug to test your software. A loopback plug is shown in Figure 10.

Finding the COM ports

Like the parallel ports, the COM port base addresses are found in the PC's BIOS table, and are standard from machine to machine. If you want to quickly check the COM port addresses for yourself, you can use the trusty old DOS debug program.

At the C:> prompt, type debug to start the program, then enter the dump command d 0:0400. The COM ports are at the beginning of the BIOS table, starting at address 0:0400, as shown in Figure 11.

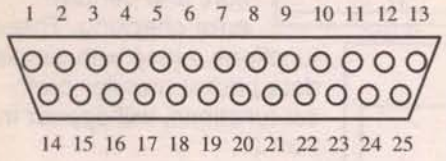
The COM port addresses start right at the beginning of the table at address 0:0400. COM1 is at address 0:0400 with data F8 03, COM2 is at address 0:0402 with data F8 02, and so on. Remember that the data appears in Little Endian format, with the least significant byte first, and the most significant byte last.

To find the actual address of COM1, take the second byte, 03, and put it in front of the first, F8. The resulting address is 03F8. Figure 12 is a table of the serial port base addresses. The values in parenthesis, such as 0x3f8 for COM1, illustrate how C needs to see hexadecimal values.

For our data link layer C

```
/*****************************************
* Filename    : datalink.h
* Description : #defines for datalink.c
* (c) 1997 Jeff Stefan
*****************************************/
/*******************
* COM port defines
*******************/
#define COM1 0x3f8
#define COM2 0x2f8
#define COM3 0x3e8
#define COM4 0x2e8

/********************
* Interrupt requests
********************/
#define IRQ_3  0x0b
#define IRQ_4  0x0c

/*********************************
* PIC address and mask values
* for INT3 and INT4.
*********************************/
#define PIC          0x21
#define EOI          0x20
#define IRQ3_MASK    0xf7
#define IRQ4_MASK    0xef
#define DISABLE_IRQ3 0x00
#define DISABLE_IRQ4 0x10

/*******************
* Register defines
*******************/
#define COM1_INT_ENAB_REG   COM1+1
#define COM1_INT_ID_REG     COM1+2
#define COM1_LINE_CTRL_REG  COM1+3
#define COM1_MODEM_CTRL_REG COM1+4
#define COM1_LINE_STAT_REG  COM1+5
#define COM1_MODEM_STAT_REG COM1+6

#define COM2_INT_ENAB_REG   COM2+1
#define COM2_INT_ID_REG     COM2+2
#define COM2_LINE_CTRL_REG  COM2+3
#define COM2_MODEM_CTRL_REG COM2+4
#define COM2_LINE_STAT_REG  COM2+5
#define COM2_MODEM_STAT_REG COM2+6

#define COM3_INT_ENAB_REG   COM3+1
#define COM3_INT_ID_REG     COM3+2
#define COM3_LINE_CTRL_REG  COM3+3
#define COM3_MODEM_CTRL_REG COM3+4
#define COM3_LINE_STAT_REG  COM3+5
#define COM3_MODEM_STAT_REG COM3+6

#define COM4_INT_ENAB_REG   COM4+1
#define COM4_INT_ID_REG     COM4+2
#define COM4_LINE_CTRL_REG  COM4+3
#define COM4_MODEM_CTRL_REG COM4+4
#define COM4_LINE_STAT_REG  COM4+5
#define COM4_MODEM_STAT_REG COM4+6

/*********************************
* Baudrate, databits, stopbits,
* and parity values
*********************************/
#define SET_DLAB        0x80

#define BAUD_300        0x180
#define BAUD_1200       0x60
#define BAUD_2400       0x30
#define BAUD_4800       0x18
#define BAUD_9600       0x0c
#define BAUD_192K       0x06

#define FIVE_DATA_BITS  0x00
#define SIX_DATA_BITS   0x01
#define SEVEN_DATA_BITS 0x02
#define EIGHT_DATA_BITS 0x03

#define ONE_STOP_BIT    0x00
#define TWO_STOP_BITS   0x04

#define NO_PARITY       0x00
#define ODD_PARITY      0x08
#define EVEN_PARITY     0x18
#define MARK            0x28
#define SPACE           0x38

#define BREAK           0x00
#define NO_BREAK        0x40

/********************
* Misc Defines
********************/
#define SET_RxRDY       0x01
#define INT_PENDING     0x01
#define CHAR_IN_UART    0x04
#define XMIT_BUF_EMPTY  0x02
#define MAX_QUEUE_SIZE  1024
```

Fig 7 DB25 Male



25-Pin	9-Pin	Signal Name
1	none	Protective Ground (see note!)
2	3	Tx Transmit Data
3	2	Rx Receive Data
4	7	RTS Request To Send
5	8	CTS Clear To Send
6	6	DSR Data Set Ready
7	5	Signal Ground
8	1	DCD Data Carrier Detect
20	4	DTR Data Terminal Ready
22	9	RI Ring Indicator

Note: NEVER use Protective Ground as a Signal Ground

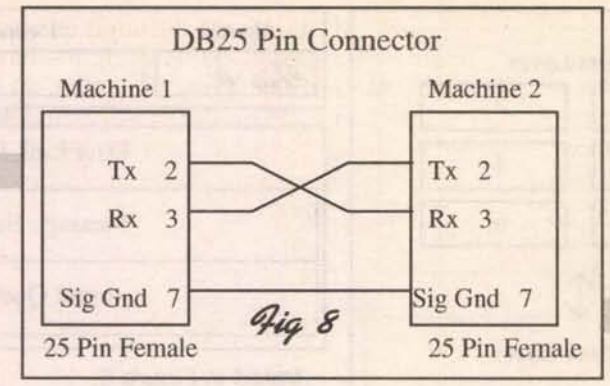


Fig 8

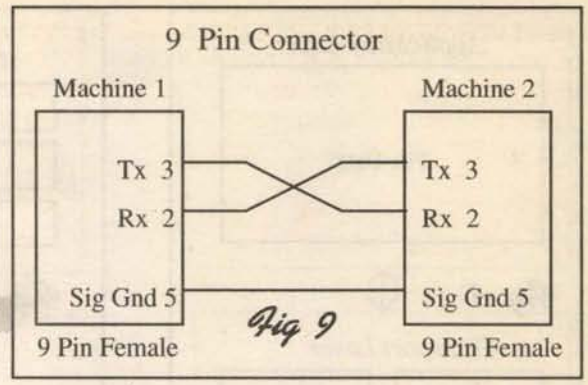


Fig 9

code, the base port addresses declarations are:

```
#define COM1 0x3f8
#define COM2 0x2f8
#define COM3 0x3e8
#define COM4 0x2e8
```

UARTS

The term UART stands for Universal Asynchronous Receiver Transmitter. The sending UART converts eight parallel bits into a serial data stream, and the receiving UART converts the incoming serial data stream back into eight parallel bits.

The most basic UART — the 8250 — is found in the early PCs, XTs, and clones. The 8250 can only receive and transmit one byte at a time before generating an interrupt. Most personal computers — from the advent of the IBM AT on up — contain a 16550 UART.

The 16550 can store up to 14 bytes before generating an interrupt, making it better suited for high throughput, multitasking applications. The 16550 can still generate an interrupt with one byte, making it compatible with the 8250. Our code uses the 8250 set-up values to remain compatible with most systems.

8250 UART Registers

UARTs are programmed via registers. The registers are accessed using input/output instructions, base addresses, and offsets to the base addresses. There are a lot of registers, and some of them share double and triple duty. If you want to do data link layer programming, you must understand the UART's registers.

Register 0

Register 0 is a triple duty register. In normal operation, Register 0 acts as the Transmit Buffer Register and Receive Buffer Register. Register 0 is at the COM base address. For COM1, its address is 3F8h. If bit 7 is set in Register 3 (Line Control Register), Register 0 also acts as the Baud Rate Divisor Latch LSB.

COM Port	Base Address	Int Num	Int Level
COM1	3F8h (0x3f8)	0x0c	4
COM2	2F8h (0x2f8)	0x0b	3
COM3	3E8h (0x3e8)	0x0c	4
COM4	2E8h (0x2e8)	0x0b	3

Fig 12

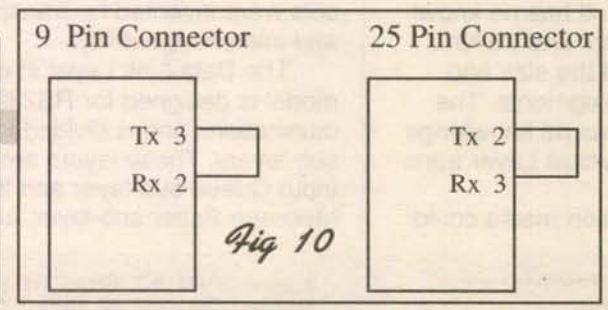


Fig 10

```
-d 0:0400
0000:0400   F8 03 F8 02 E8 03 E8 02- 00 00 00 00 00 00 00 00
```

Fig 11

Register 1

Register 1 is the Interrupt Enable Register. Register 1 uses only four bits (bit 0 through bit 3) to enable interrupts. Bits 4 through 7 must be set to zero. Register 1 is addressed by the base port address plus one. The data link C code declaration for Register 1 is #define COM1_INT_ENAB_REG COM1 + 1.

This assumes that COM1 is the active serial port. If COM2 is used, the #define must be changed to #define COM2_INT_ENAB_REG COM2 + 1. The same holds for COM3 and COM4. This method allows you to keep your addresses cleanly separated if you use more than one COM port at a time.

The bit that the data link layer code depends on is Bit 0, the RxRDY bit. If this bit is set to 1, then the UART will generate an interrupt whenever a character enters the Receive Buffer. To set bit 1, the C code uses #define RxRDY_BIT 0x01.

Register 2

Register 2 is the Interrupt Identification Register. While Register 1 allows the interrupts to happen, Register 2 indicates which interrupts have occurred.

Register 2 can be examined in two steps. Step 1 is to examine Bit 0, the Interrupt Pending bit. If an interrupt occurred, then Bit 0 will be set to one, and the software needs to examine Bits 1 and 2.

Bits 1 and 2 tell the software what kind of interrupt occurred. Bit 0 just tells you that an interrupt occurred. If Bit 0 is zero, then an interrupt did not occur, and there's no point in looking at the register any further. The address of Register 2 in the C code is: #define COM1_INT_ID_REG COM1 + 2.

By examining Bit 1 and Bit 2 together, the data link layer can determine if any of the RS232 lines have changed (Modem Status Change), if there are no characters to transmit (Transmit Buffer Empty), if a character is waiting to be read (Char is Receive Buffer), or if an error condition or break occurred (Error or Break).

Register 3

Register 3 is the Line Control Register. This register is used at initialization time to correctly set up the number of data bits, stop bits, type of parity checking used, and whether to recognize a break signal or not. Register 3 is addressed as COM1_LINE_CTRL_REG COM1 + 3.

Register 4

Register 4 is the Modem Control Register. This register controls the hardware handshaking signals RTS and DTS. It also controls the General Purpose Outputs. GP02 must be set for the UART's interrupt to work on the PC.

Register 5

Register 5 is the Line Status Register. This register is used to monitor activity on the transmission lines. Bit 0 — the RxRDY bit — can be used by an interrupt service routine to see if a character waiting to be read in the receive buffer.

Register 6

Register 6 is the Modem Status Register. This register monitors the status of handshaking lines.

Register 7

Register 7 is the Scratch Pad Register. This is a register that is not used, and should not be used, since some of the early 8250s did not have this register.

Register 8

Register 8 is the Least Significant Byte Baud Rate Divisor Latch. This is really at the address of Register 0.

Register 9

Register 9 is the Most Significant Byte Baud Rate Divisor Latch. This is really at the address of Register 1.

Interrupt Service Routines

When a serial interrupt occurs, an interrupt handler must take temporary control of program execution and process the data associated with the interrupt. Just setting up the UART registers will do nothing. You need an interrupt service routine to do the work. Our interrupt service routine, or interrupt handler as they are sometimes called, is written in C.

Most interrupt service routines are written in assembly language, including ones written for serial communications. Why is our ISR written in C? There are two reasons. The first reason is that most C compilers supply the tools to write interrupt level function calls, so why not use them? Everyone reading and using the code in this article may not have access to an assembler.

The second reason is that it's easier to write interrupt service routines in C than it is in assembly. Granted, there's more overhead in a C interrupt service routine, and, in a lot of cases, there's no substitute for using assembly language. When

I need to write an interrupt service routine I first write it in C, then see how it performs. If it works as expected, I keep it. If it doesn't perform well, then I rewrite it in assembly language.

Turbo C/C++ supports interrupt level functions. To write an ISR, just insert the keyword interrupt in your function declaration. Our interrupt handler is declared as void interrupt DataLinkISR(void); the function doesn't return a value, so it is of type void. The keyword interrupt is placed in between the function return type and function name. There are no parameters passed through the function, hence the void in the argument list.

There are a few additional

```
/*********************************************************
* Filename    : datalink.c
* Description : Data Link Layer serial communications
*               routines.
* Note: Compile with Large or Small Memory Model
*
* (c) 1996 Jeff Stefan
*
*********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <conio.h>

#include "datalink.h"

#define ESCAPE 0x1b
#define DEBUG

/*********************
* Function prototypes
*********************/
void interrupt DataLinkISR(void);
void interrupt (*OldISR)(void);

int InitCom1(void);
int DisableCom1(void);
int CheckBytes(void);
int ReadByte(void);
void WriteByte(unsigned char);

/************************
* Input queue variables
************************/
unsigned char InputQueue[MAX_QUEUE_SIZE];
unsigned char *HeadPtr,*TailPtr,*StartOfQueue,*EndOfQueue;
unsigned char PicVal=0x00;

/*******************************
* Input Message Buffer and index
*******************************/
unsigned char InputMsgBuff[MAX_QUEUE_SIZE];
int BufIdx=0;

#ifdef DEBUG
  unsigned char Dbg;
  unsigned int  DbgCnt=0;
#endif

int main()
{
int InChar=0x00;

  /*****************************************************************
  * Init COM1 to 9600 baud, 8 data bits, 1 stop bit, and no parity
  *****************************************************************/
  InitCom1();

  /*******************************
  * Init the input queue pointers
  *******************************/
  HeadPtr    = TailPtr = StartOfQueue = InputQueue;
  EndOfQueue = StartOfQueue + MAX_QUEUE_SIZE;

  /*****************
  * do terminal
  *****************/
  while(InChar != ESCAPE)
  {
    if(kbhit())
    {
      InChar = getch();
      WriteByte(InChar);
    }
    else
    if(CheckBytes())
    {
      ReadByte();
    }
  }

  DisableCom1();

#ifdef DEBUG
  printf("Debug Count = %d\n",DbgCnt);
#endif

return(0);
}

/*********************************************************
* InitComPort : initializes COM port.
*
*********************************************************/
int InitCom1()
{
unsigned char PortData;

  /*************************************
  * Set Line Control Register DLAB bit 7
  * to enable baudate initialization.
  *************************************/
  PortData = inportb(COM1_LINE_CTRL_REG);
  outportb(COM1_LINE_CTRL_REG, SET_DLAB | PortData);

#ifdef DEBUG
  Dbg = inportb(COM1_LINE_CTRL_REG);
#endif

  /*************************************
  * Write baudrate value to Reg0 and Reg1
  *************************************/
  outport(COM1,BAUD_192K);

  /*********************************************
  * save the old interrupt service routine at INT4
  *********************************************/
  OldISR = getvect(IRQ_4);

  /*************************************
  * use setvect() to point to com ISR
  *************************************/
  setvect(IRQ_4,DataLinkISR);

  /*************************************
  * Setup Line Control Reg for eight data
  * bits, one stop bit, and no parity.
  *************************************/
  outportb(COM1_LINE_CTRL_REG,EIGHT_DATA_BITS |
        ONE_STOP_BIT | NO_PARITY);

#ifdef DEBUG
  Dbg = inportb(COM1_LINE_CTRL_REG);
#endif

  /*****************************************
  * Set the Modem Control Register
  *****************************************/
  outportb(COM1_MODEM_CTRL_REG,0x0b);

  /*********************************************
  * set Int Enable Reg for rcv interrupt only
  *********************************************/
  outportb(COM1_INT_ENAB_REG,SET_RxRDY);

#ifdef DEBUG
  Dbg = inportb(COM1_INT_ENAB_REG);
#endif

  /**********************
  * set up PIC for INT4
  **********************/
  PicVal = inportb(PIC);
  outportb(PIC,PicVal & IRQ4_MASK);

#ifdef DEBUG
  Dbg = inportb(PIC);
#endif

  /*************************************
  * read char from port to flush input
  *************************************/
#ifdef DEBUG
  Dbg = inportb(0x3f8);
#endif

#ifdef DEBUG
  Dbg = inportb(0x3f8);
  Dbg = inportb(0x3f9);  // int enab
  Dbg = inportb(0x3fb);  // line ctrl
  Dbg = inportb(0x3fc);  // modem ctr
  Dbg = inportb(0x3fd);  // line status
  Dbg = inportb(0x3fe);  // modem status
  Dbg = inportb(0x21);   // PIC

#endif

  return(0);
}

/*********************************************************
* DisableCom1 :  disables COM1 port.
*
*********************************************************/
int DisableCom1()
{
  /*****************************************
  * restore old interrupt service routine
  *****************************************/
  setvect(IRQ_4,OldISR);

  /************************
  * restore old PIC value
  ************************/
  PicVal = inportb(PIC);
  outportb(PIC,PicVal | DISABLE_IRQ4);

#ifdef DEBUG
  Dbg = inportb(PIC);
#endif

  return(0);
}

/*********************************************************
* DataLinkISR : serial port interrupt service routine.
*
*********************************************************/
void interrupt DataLinkISR(void)
{
unsigned char InChar;

 enable();

 InChar = inportb(COM1);
 if(TailPtr == EndOfQueue)
 {
   TailPtr = StartOfQueue;
   *TailPtr++ = InChar;
 }
 else
 {
   *TailPtr++ = InChar;
 }

#ifdef DEBUG
  DbgCnt+=1;
#endif

 /*****************************
 *  re-enable PIC interrupts
 *****************************/
 outportb(EOI,EOI);

}

/*********************************************************
* CheckBytes : checks if queue contains unread chars. Returns
*              zero in no bytes in queue, else returns number
*              of chars in queue.
*********************************************************/
int CheckBytes(void)
{
  /*****************************************
  * check if head pointer == tail pointer
  * if ==, no chars are in buffer: return(0);
  * num chars = abs diff from head to tail
  * return num chars.
  *****************************************/
  if(HeadPtr - TailPtr)
  {
    return(abs(HeadPtr - TailPtr));
  }
  else
  {
  return(0);
  }
}

/*********************************************************
* ReadByte : reads a byte from the queue
*
*********************************************************/
int ReadByte(void)
{
  /*****************************************
  * check if head pointer == tail pointer
  * if ==, no chars are in buffer: return(0);
  *****************************************/
  if(HeadPtr >= TailPtr)
  {
    HeadPtr = StartOfQueue;
    return(0);
  }
  else
  {
  printf("%c",*HeadPtr);
  HeadPtr++;
  }
 return(0);
}

/*********************************************************
* WriteByte : writes a byte out the serial port
*
*********************************************************/
void WriteByte(unsigned char CharToSend)
{
  /*************************************
  * Wait for the transmit flag to clear
  *************************************/
  while((inportb(COM1_LINE_STAT_REG) & 0x20) == 0)
  {
    ;
  }
  /*****************
  * Output the byte
  *****************/
  outportb (COM1,CharToSend);
}
```

Register 0

Transmit Buffer Register / Receive Buffer Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Register 1

Interrupt Enable Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	Modem Status	Error / Break	TBE	RxRDY

Bit 0 RxRDY When set to 1, generates interrupt when char is in Receive Buffer Register

Bit 1 TBE When set to 1, generates interrupt when char is ready to xmit in Transmit Buffer Register

Bit 2 Error/Break When set to 1, generates interrupt whenever a parity error, framing error, break, or overrun

Bit 3 Modem Status When set to 1, generates interrupt whenever any RS232 input line changes

Bit 4 - 7 Unused (always set these bits to 0)

Register 2

Interrupt Identification Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	Interrupt Type Table		Int Pend

Bit 0 If = 0, interrupt occured. If = 1, interrupt occured

Bit 1-2 Interrupt Type Table if = 0.

Bit 2	Bit 1	Type
0	0	Modem Status Change
0	1	Transmit Buffer Empty
1	0	Char in Receive Buffer
1	1	Error or Break

Bit 3 - 7 Always 0

Register 3

Line Control Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Bit 0 - 1 Number of Data Bits

Bit 0	Bit 1	
0	0	5 data bits
0	1	6 data bits
1	0	7 data bits
1	1	8 data bits

Bit 2 If 0, 1 stop bit, if 1, 2 stop bits

Bit 3 - 5 Parity

Bit 5	Bit 4	Bit 3	
0	0	0	NO Parity
0	0	1	ODD Parity
0	1	1	EVEN Parity
1	0	1	MARK
1	1	1	SPACE

Bit 6 1 = BREAK, 0 = No BREAK

Bit 7 Divisor Latch Access Bit (DLAB)
If = 0, Normal Operation
If = 1, Baud Rate Divisor Latch

things you need to do to ensure that your ISR works correctly. You need to turn off the Stack Warning option, and you need to make sure that the Register Variables option is set to none. To set these values, select Options from the top menu bar. You can set these options in the Linker and Optimizations menu selections.

One little pitfall using an interrupt service routine with an Intel processor is that interrupts are automatically disabled when inside an ISR. Sometimes this is okay, if the code is critical and cannot be pre-empted by another ISR. Usually, interrupts should be immediately re-enabled inside an ISR. This is done by calling the enable( ) macro. This macro is defined as asm sti, which is the assembly language instruction to enable interrupts. Our ISR immediately upon entry calls enable( ).

One last chip must be set up before an ISR can work. This chip is the 8259 Programmable Interrupt Controller, or PIC. There are two PICs in a PC and each of the interrupt requests — IRQ0 through IRQ15 — are wired to the PICs.

In order to get an ISR to run, a bit corresponding to the ISR must be set in the PIC. This is done by writing a mask value to address 0x20. One common mistake when writing ISRs, especially in embedded systems programming, is to forget to set the PIC.

Another common mistake is to forget to reset the PIC after an interrupt occurs. This is done by writing an End of Interrupt, or EOI to address 0x21. If this isn't done, then your interrupt service routine will run one time, and one time only. Resetting the PIC is done within the ISR. In our ISR, this is done by the outportb(PIC,PIC) instruction.

If this seems like a lot of complicated grunt work, you're right. But it's worth taking the time to become familiar with the PC's hardware. It aids in your mastery of the machine. It allows you to control the computer, and not vice-versa. Here's the checklist for installing an interrupt service routine:

1. Write the baudrate values to the UART.
2. Save the old ISR at the interrupt vector and install the new COM ISR.
3. Set up the UART's Line Control Register to the desired control values (data bits, parity, etc.).
4. Set the UART's Interrupt Enable Register to enable the receive interrupt.
5. Set the bit for the ISR in the PIC register.

Low-Level Functions

The Input Queue sub-layer functions are InitCom1( ), DisableCom1(), CheckBytes( ), ReadByte( ), WriteByte( ), and the interrupt service routine DataLinkISR( ). The key data structure is the array InputQueue[MAX_QUEUE_SIZE], and its associated pointers. The function InitCom1( ) initializes COM1 to 19.2K baud, 8 data bits, 1 stop bit, and no parity.

A function to initialize COM2 can be easily written by substituting the COM2 definitions in place of the COM1 definitions.

DisableCom1( ) restores the original interrupt service routine and restores the original PIC value. The CheckByte( ) function checks to see if any characters are in the InputQueue. The ReadByte( ) function extracts a character from the input queue using the *HeadPtr. The WriteByte( ) function sends a character out the serial port.

The InputQueue array pointers are initialized by setting HeadPtr = TailPtr = InputQueue. This sets the pointer addresses to the start of the InputQueue array. The EndOfQueue pointer is initialized to StartOfQueue + MAX_QUEUE_SIZE, effectively setting EndOfQueue to the ending address of InputQueue.

The function that does most of the work is the com port initialization function InitCom1( ). This function sets up the UART registers, saves the original ISR, installs the new ISR, and sets up the PIC to recognize the newly installed ISR. Read the comments in the code listing for InitCom1( ) for step by step details.

You will notice that a debug variable, named Dbg, appears frequently in the function and throughout the code. When I develop interrupt service routines and code that involves accessing and controlling the PC's hardware, I always use one or two debug variables to examine the contents of the hardware's registers, especially just after writing to them. That way it's easy to verify that the registers are set correctly when stepping through the code in a debugger. This is especially handy when trying to set the PIC to recognize a newly installed ISR.

The other debug variable used is DbgCnt, which resides in the DataLinkISR( ) interrupt service routine. This is the easiest way to see if your interrupt service routine is being called. Every time the ISR runs, DbgCnt increments by one. When the program terminates, the value of DbgCnt is printed, telling you how many times the ISR was called.

If DbgCnt is zero, then your ISR never ran, so it may not be installed at the right address or the PIC may be set up incorrectly. If DgbCnt is one, then your ISR ran one time only, so you probably didn't re-enable the PIC interrupt correctly. It's amazing what a simple counter can tell you!

The heart of the Input Queue sub-layer is the DataLinkISR( ) function. This is the interrupt service routine that reads characters from the physical layer, which is the UARTS receive register. Here's the ISR code:

```
void interrupt DataLinkISR(void)
        {
 unsigned char InChar;

enable( );
 InChar = inportb(COM1);
if(TailPtr == EndOfQueue)
{
TailPtr = StartOfQueue;
*TailPtr = InChar;
}
else
{
```

Register 4
Modem Control Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	Local Loop Back	GP02	GP01	RTS	DTR

Bit 0 DTR (Data Terminal Ready) If set to 1, DTR line goes high, if set to 0, DTR line goes low

Bit 1 RTS (Request to Send) If set to 1, RTS line goes high, if set to 0, RTS line goes low

Bit 2 GP01 (General Purpose Output 1) User definable output

Bit 3 GP02 (General Purpose Output 2) User definable output

Bit 4 LL (Local Loopback) If set to 1, enables loopback function

Bit 5 - 7 Always 0

Register 5
Line Status Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0							

Bit 0 RxRDY If = 1, char is ready to read in buffer (Register 0). If = 0, no char in buffer

Bit 1 If = 1, overrun error occured

Bit 2 If = 1, parity error occured

Bit 3 If = 1, framing error occured

Bit 4 If = 1, BREAK detected

Bit 5 If = 1, transmit buffer is empty and ready for another character

Bit 6 If = 1, transmit buffer is flushed

Bit 7 Always 0

Register 6
Modem Status Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CD state	RI state	DSR state	CTS state	CD	RI	DSR	CTS

Bit 0 If = 1, CTS line changed since last read

Bit 1 If = 1, DSR line changed since last read

Bit 2 If = 1, RI line changed since last read

Bit 3 If = 1, CD line changed since last read

Bit 4 Indicates state of CTS line

Bit 5 Indicates state of DSR line

Bit 6 Indicates state of RI line

Bit 7 Indicates state of CD line

Register 7
Scratchpad Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

General pupose scratchpad register (not on early 8250s)

Register 8 (Resister 0)
LSB Baud Rate Divisor Latch

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

This is the Least Significant Byte value used to set the 8250 baudrate.
Register 8 is accessed by setting bit 7 of the Line Control Register (Register 3) to 1.

Register 9 (Resister 1)
MSB Baud Rate Divisor Latch

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

This is the Most Significant Byte value used to set the 8250 baudrate.
Register 9 is accessed by setting bit 7 of the Line Control Register (Register 3) to 1.

Baud Rate Divisor Latch Values

Baud Rate	Latch Value
300	384
1200	96
2400	48
4800	24
9600	12
19.2K	6

```
*TailPtr++ = InChar;
}
outportb(EOI,EOI);
}
```

As you can see, the ISR is short and relatively simple. Upon entry, interrupts are re-enabled by calling enable( ), then a character is read from COM1. Before the character is placed in the queue, the TailPtr is tested to see if it's at the end of the queue. If it is, then the TailPtr is reset to StartOfQueue and only then is the character placed in the queue. The TailPtr then advances to the next queue position. Finally, the 8259 PIC is re-enabled by the outportb(EOI,EOI) function.

Characters are extracted by the ReadByte( ) function, which is very simple. The code for ReadByte( ) is:

```
int ReadByte(void)
{
if (HeadPtr >= TailPtr)
{
HeadPtr = StartOfQueue;
return(0);
}
else
{
printf("%c", *HeadPtr);
HeadPtr++;
}
return(0);
}
```

On entry, the HeadPtr is tested against the TailPtr and is set to the StartOfQueue if it's out of bounds. The extracted character is printed and the HeadPtr is advanced to the next character in the InputQueue array.

The CheckByte( ) function calculates the difference between the *HeadPtr and *TailPtr to determine the number of characters in the buffer. The WriteByte( ) function reads the Line Status Register (Register 5) and checks if the transmit buffer is empty by performing a logical and with Bit 5. If Bit 5 is set to one, then the transmit buffer is ready to send another character. The outportb(COM1, CharToSend) function sends the new character out the serial port.

That's it for the lowest level of the Data Link Layer. With these functions in place, frames can be built and detected, and protocols can be developed to support high level functions such as file transfer, remote procedure calls, and client/server applications.

These simple functions are the mortar that hold virtual machines together. The example program, datalink.c, and its associated header file datalink.h, allows characters to be received and transmitted between machines. You are encouraged to experiment with the code. The code is left intentionally sparse so you can easily modify it.

In Part 2, we will build frames from the InputQueue based on the read/write functions, evaluate various error checking methods such as checksums and CRCs, and build a simple protocol that you can use to transfer data packets from machine to machine. **NV**

Serial Communications and Protocol Stacks:

Why All The Layers?

Part 2

by Jeff Stefan

In Part 1 of this article, we examined the structure of protocol stacks and developed a set of low-level interrupt driven RS232 communications functions. These functions serve the Input Queue level sub-level of the Data Link Layer.

In Part 2, we will build the Message Buffer service functions and develop a Data Link protocol to transfer data packets from machine to machine.

A Little History: OSI versus TCP/IP

In the mid-70s, designers were creating proprietary computer networks. Unfortunately, none of these proprietary networks could talk reliably to each other. It soon became apparent that some kind of communications standards, or protocols, were needed to exploit the full potential of networked systems. An ISO (International Standards Organization) committee was quickly formed in 1978 to create a set of standards called Open Systems Interconnection, thus coining the acronym OSI. After about 18 months, the Reference Model for Open Systems Interconnection was complete. The result is the Seven Layer Model, as illustrated in Figure 1.

The developers of OSI were concerned primarily with the Reference Model architecture, and less concerned with protocols. The OSI architecture has deep roots in telecommunications, and was not tailored specifically for computer-to-computer communications. Figure 2 contains an overview of each of seven layers.

While political battles occurred over OSI, the government was quietly concerned about how systems would continue to communicate in the event of a nuclear war or other catastrophic national disaster. The Department of Defense sponsored a research network called the ARPANET, which stands for the Advanced Research Projects Agency Network.

The ARPANET rapidly expanded throughout universities and government agencies. ARPANET developers were more concerned with actual protocols rather than focusing on a formal architecture. From their research came two primary protocols: TCP, which stands for Transport Control Protocol; and IP, which stands for Internet Protocol. From these protocols an architecture emerged, called the TCP/IP Reference Model. Computer networks were now communicating reliably, and over great distances. Real work was being done. The TCP/IP reference model is shown in Figure 3.

The TCP/IP Reference Model contains less layers than the OSI model. That's because the OSI Session and Presentation Layers were inserted into the model to be compatible with IBM's System Network Architecture, or SNA. These layers are seldom used in practice, so the TCP/IP designers left them out. The TCP and UDP protocols are located in the Transport Layer, and the IP is located in the Network Layer. UDP stands for User Datagram Protocol. From this model, the Internet was built.

Frame Format

Our protocol stack model contains four layers, as illustrated in Figure 4.

Our focus is on the Data Link Layer. The Data Link Layer is divided into two sub-layers, called the Input Queue and the Message Buffer. The functions developed in Part 1 of this article write bytes and read bytes to and from a serial port. These functions provide the lowest level of services in the Data Link Layer. The Message Buffer layer functions build on these lower level services to send and receive frames. The functions associated with each of the sub-layers are illustrated in Figure 5.

The raw byte stream provided by the Input Queue sub-layer isn't good enough for reliable machine-to-machine communications. The byte stream needs to be assembled into meaningful packets, or frames. By building frames, the upper layer functions can check the integrity of the data transmission and reliably extract meaningful information from the frame. The frame format we will use is shown in Figure 6.

The frame is constructed from three components: the Header, Payload Data, and Trailer. The frame is defined by a set of three C structures as shown in Figure 7.

The Header and Data segments are defined in separate structures, then merged into one structure called FRAME. This makes it easier to experiment with the header and data segments. It also makes it easier to write functions that act only on the header, and only on the data.

Checksums

The error checking method we will employ is a simple checksum. Here's how it works. As the sender builds a packet, the bytes are added up, then every bit is inverted. The inverted sum is placed in the last byte. When the receiver gets the packet, it adds the bytes, then performs an exclusive "or" operation using the bytes it added and the inverted sum in the packet. If the result is FFh (all ones), then the packet is good. If the packet is not all ones, then an error occurred and it must be re-transmitted. An example of how this works is shown in Figure 8.

How is the sum inverted by the Sender? It's easy in C. All you have to do is use the one's complement operator, which is the tilde. Once the bytes are added, the line of code to accomplish this is:

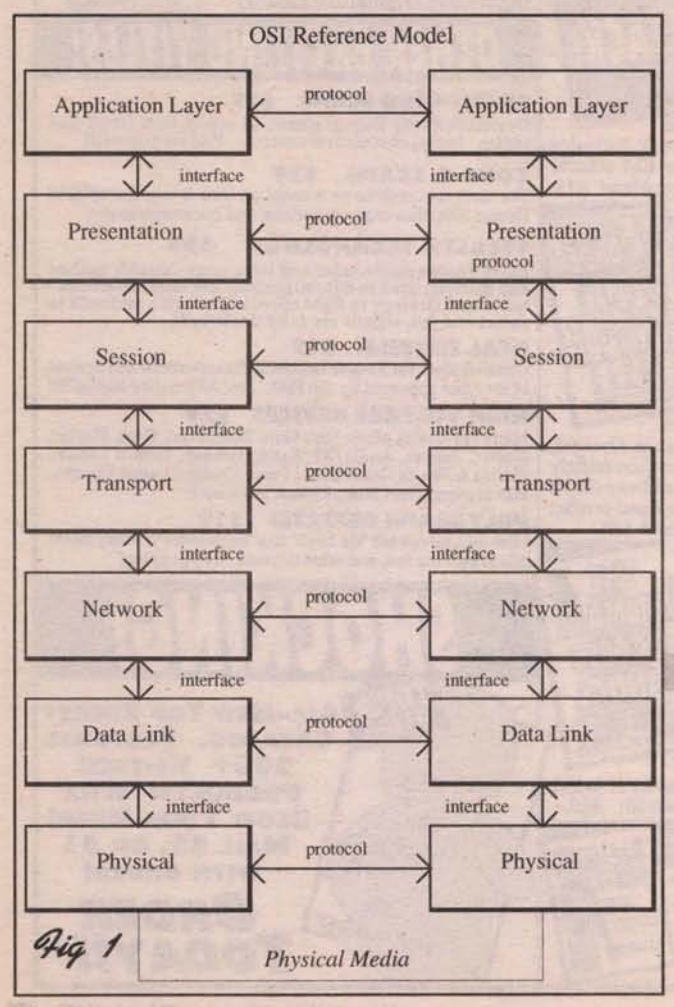


Fig 1

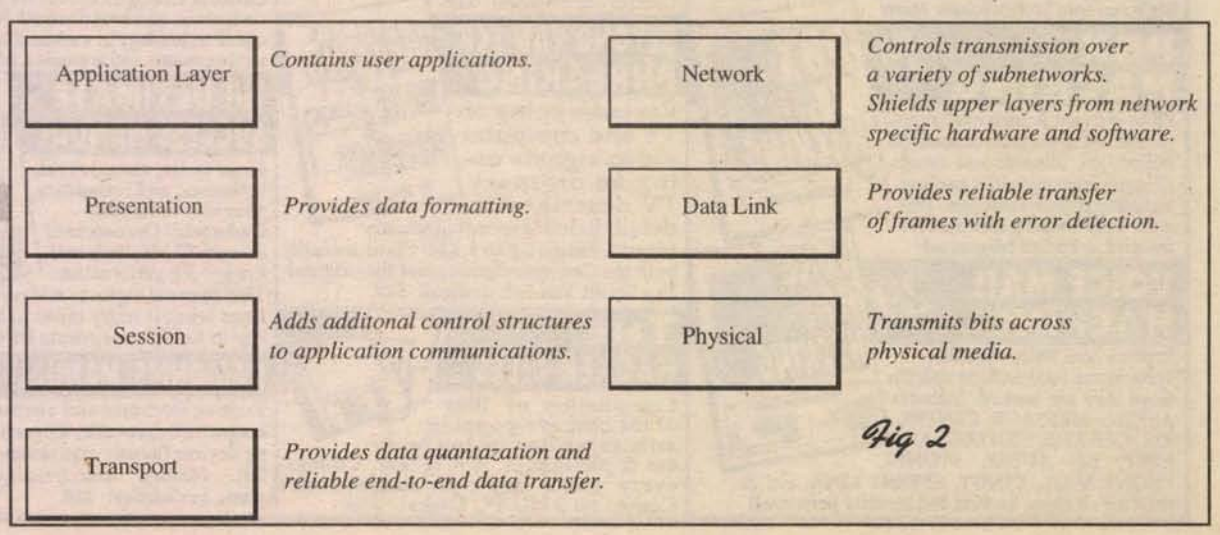

Application Layer
Contains user applications.
Presentation
Provides data formatting.
Session
Adds additonal control structures to application communications.
Transport
Provides data quantazation and reliable end-to-end data transfer.
Network
Controls transmission over a variety of subnetworks. Shields upper layers from network specific hardware and software.
Data Link
Provides reliable transfer of frames with error detection.
Physical
Transmits bits across physical media.

Fig 2

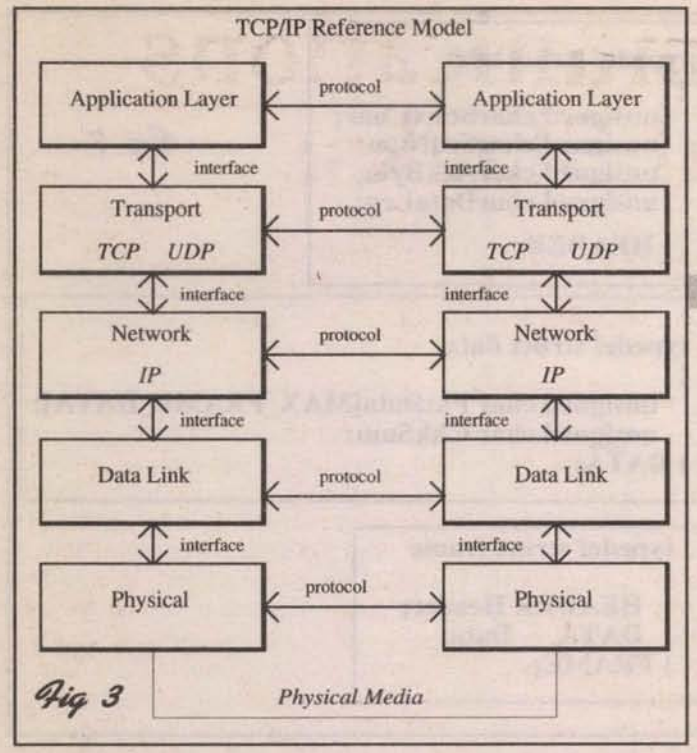


Fig 3

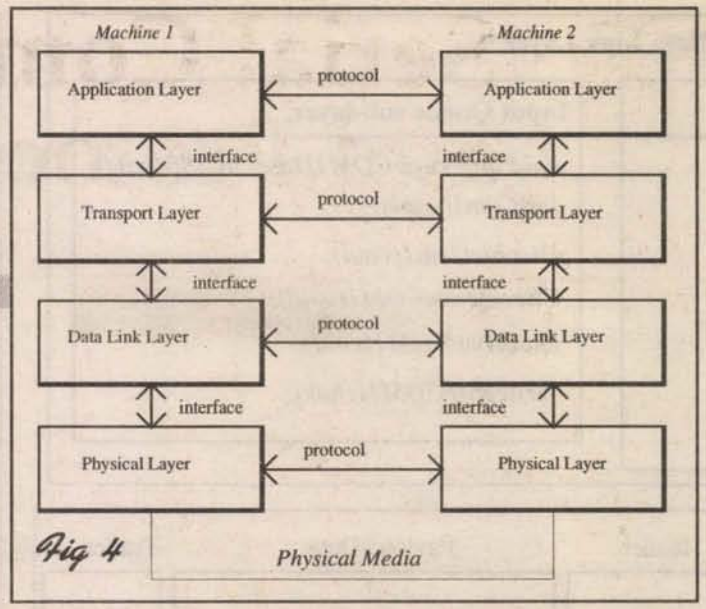


Fig 4

Chksum = ~(Chksum);

Cyclic Redundancy Checks

Cyclic Redundancy Checks, or CRCs, are the most effective way of detecting transmission packet errors. CRCs are commonly implemented in hardware. CRCs are based on frame check sequences, which can be generated from data packets. The frame check sequence is sometimes referred to as a polynomial generator. Frames and frame sequences are viewed as polynomials. For example, the frame 10110111 is represented as x7 + x5 + x3 + x2 + x1 + 1. The frame check sequence 10101 is represented as x4 + x2 + 1.

Here's how CRCs work: The frame is divided by the frame check sequence, and the remainder tacked on to the end of the frame. The frame is then transmitted. Both the sender and receiver agree in advance on the frame check sequence. When the receiver gets the frame, it divides the frame by the frame check sequence. If the result is zero, then the packet contains no errors. There are three commonly used frame check sequences, as shown in Figure 9.

Packet Structure

In this section, we'll construct the transmission frames, or packets. For our purposes, the term frame and packet are interchangeable. The function used to build a packet (or frame) is called BuildFrame (unsigned char *DataToSend). This function performs eight clearly defined steps. These steps are:

1. Increment the packet sequencecounter.
2. Load the packet data (that's the DataToSend character array passed to the function).
3. Set the ACK byte to zero.
4. Calculate the packet length.
5. Load the length byte.
6. Load the start character ':'.
7. Calculate the packet checksum.
8. Load the checksum byte.

These steps are short and simple. Refer to BuildFrame (DataToSend) in the program listing for DL1.C for the details. After this function is called, the Frame structure is loaded and ready for transmission. Figure 10 illustrates a completed frame with data 12345 loaded into the Frame.Data.PktData array. These are the bytes you will see if you build and transmit this frame.

As you build frames, the Sequence Number will increment each time, allowing the receiving side to track the frames. Building and sending frames is easier than detecting and processing frames.

Once the frame is sent, the receiver must extract the frame from the message buffer. Remember our data link layer is broken into two subsections: the Input Queue, which contains raw characters received from the physical layer; and the Message Buffer, which contains packets or frames.

The Protocol

The protocol presented here is a simple Stop and Wait protocol. A Stop and Wait protocol sends a frame, then waits for a response packet. No other communication occurs when the sender is waiting for a response packet. The commu-

```
/*****************************************************
* Filename    : dl1.h
* Description : Data Link Layer serial communi-
* cations header file.
*
* Notes: This file contains the #defines and
* data structures used to create, receive, and
* transmit frames.
*
*
* (c) 1997 Jeff Stefan
*
*****************************************************
*/
#define TIME_OUT_VAL  1000

#define MAX_FRAME_DATA  256
#define MAX_BUFFER_SIZE 1024
#define HEADER_SIZE 4
#define ACK 0x06
#define NAK 0x15

/*********************
* Header structure
*********************/
typedef struct hdr
{
  unsigned char StartChar;
  unsigned char SeqNum;
  unsigned char FrameLen;
  unsigned char AckByte;
}HEADER;

/*********************
* Data structure
*********************/
typedef struct data
{
  unsigned char PktData[MAX_FRAME_DATA];
  unsigned char ChkSum;
}DATA;

/*********************************************
* Frame structure built from HEADER and
DATA
*********************************************/
typedef struct frame
{
 HEADER Header;
 DATA  Data;
}FRAME;
```

```
/*****************************************
* Filename    : datalink.h
* Description : #defines for datalink.c
*
*****************************************/

/*******************
* COM port defines
*******************/
#define COM1 0x3f8
#define COM2 0x2f8
#define COM3 0x3e8
#define COM4 0x2e8

/*********************
* Interrupt requests
*********************/
#define IRQ_3  0x0b
#define IRQ_4  0x0c

/*******************************
* PIC address and mask values
* for INT3 and INT4.
*******************************/
#define PIC         0x21
#define EOI         0x20
#define IRQ3_MASK    0xf7
#define IRQ4_MASK    0xef
#define DISABLE_IRQ3 0x00
#define DISABLE_IRQ4 0x10

/*******************
* Register defines
*******************/
#define COM1_INT_ENAB_REG   COM1+1
#define COM1_INT_ID_REG     COM1+2
#define COM1_LINE_CTRL_REG  COM1+3
#define COM1_MODEM_CTRL_REG COM1+4
#define COM1_LINE_STAT_REG  COM1+5
#define COM1_MODEM_STAT_REG COM1+6

#define COM2_INT_ENAB_REG   COM2+1
#define COM2_INT_ID_REG     COM2+2
#define COM2_LINE_CTRL_REG  COM2+3
#define COM2_MODEM_CTRL_REG COM2+4
#define COM2_LINE_STAT_REG  COM2+5
#define COM2_MODEM_STAT_REG COM2+6

#define COM3_INT_ENAB_REG   COM3+1
#define COM3_INT_ID_REG     COM3+2
#define COM3_LINE_CTRL_REG  COM3+3
#define COM3_MODEM_CTRL_REG COM3+4
#define COM3_LINE_STAT_REG  COM3+5
#define COM3_MODEM_STAT_REG COM3+6

#define COM4_INT_ENAB_REG   COM4+1
#define COM4_INT_ID_REG     COM4+2
#define COM4_LINE_CTRL_REG  COM4+3
#define COM4_MODEM_CTRL_REG COM4+4
#define COM4_LINE_STAT_REG  COM4+5
#define COM4_MODEM_STAT_REG COM4+6

/*******************************
* Baudrate, databits, stopbits,
* and parity values
*******************************/
#define SET_DLAB        0x80

#define BAUD_300        0x180
#define BAUD_1200       0x60
#define BAUD_2400       0x30
#define BAUD_4800       0x18
#define BAUD_9600       0x0c
#define BAUD_192K       0x06

#define FIVE_DATA_BITS   0x00
#define SIX_DATA_BITS    0x01
#define SEVEN_DATA_BITS  0x02
#define EIGHT_DATA_BITS  0x03

#define ONE_STOP_BIT     0x00
#define TWO_STOP_BITS    0x04

#define NO_PARITY        0x00
#define ODD_PARITY       0x08
#define EVEN_PARITY      0x18
#define MARK             0x28
#define SPACE            0x38

#define BREAK            0x00
#define NO_BREAK         0x40

/*******************
* Misc Defines
*******************/
#define SET_RxRDY       0x01
#define INT_PENDING     0x01
#define CHAR_IN_UART    0x04
#define XMIT_BUF_EMPTY  0x02
#define MAX_QUEUE_SIZE  1024
```

Program listing from Part 1

Fig 5

Data Link Layer

Message Buffer sub-layer	Input Queue sub-layer
*void BuildFrame(char *);*	*void interrupt COM1DataLinkISR(void);*
void GetFrame(void);	*InitCom1(void);*
void PutFrame(void);	*DisableCom1(void);*
int CheckFrame(int);	*CheckBytesCOM1(void);*
void FlushQueue(void);	*ReadByteCOM1(void);*
void SendAckResponse(void);	*WriteByteCOM1(char);*
void SendNakResponse(void);	

Fig 7

```
typedef struct hdr
{
   unsigned charStartChar;
   unsigned charSeqNum;
   unsigned charAckByte;
   unsigned charDataLen;
 } HEADER;
```

```
typedef struct data
{
   unsigned char PktData[MAX_FRAME_DATA];
   unsigned char ChkSum;
} DATA;
```

```
typedef struct frame
{
   HEADER Header;
   DATA    Data;
 } FRAME;
```

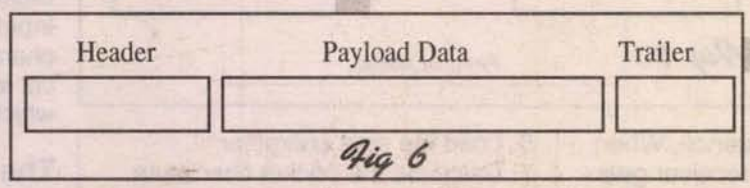


Fig 6

nications flow is illustrated in Figure 11.

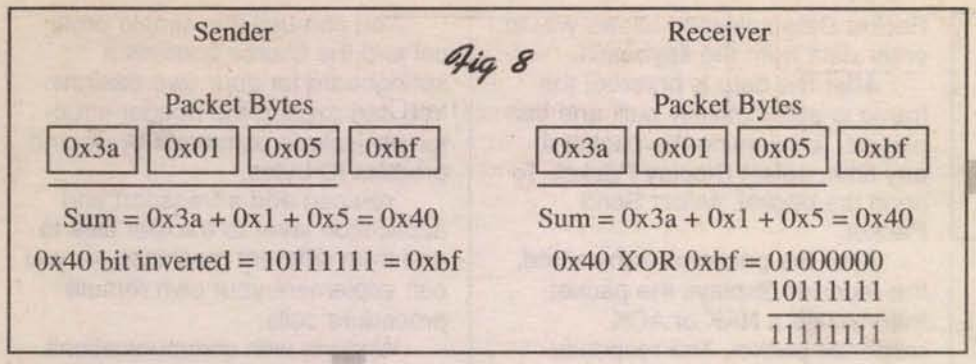
The sender builds a frame and transmits it to the receiver, then waits. The receiver checks the incoming frame for errors then issues an appropriate response packet. If the received frame is bad, then the receiver issues a NAK response packet. This means that the packet was corrupted on the way to the receiver, and that the same packet must be re-transmitted by the sender with the same sequence number. If the receiver issues an ACK response packet, then the sender is free to increment the sequence number and send the next frame.

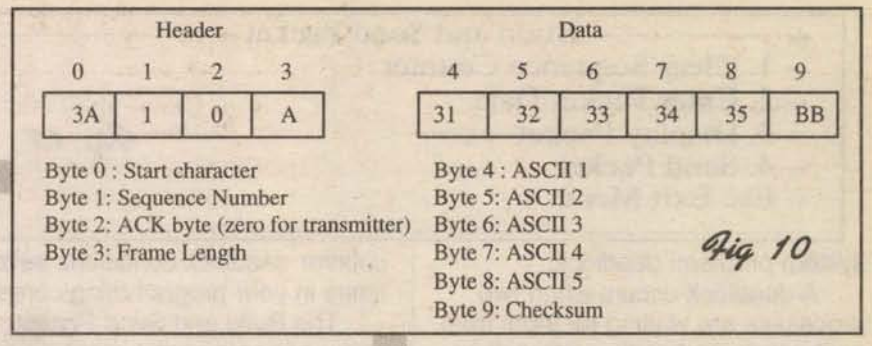
This protocol is far from perfect, but it's a good place to start. Protocols start getting complicated in a hurry, and it's a good idea to start with something simple that works.

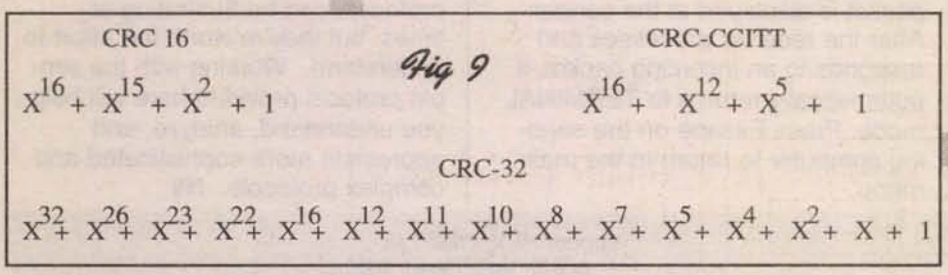
The Program

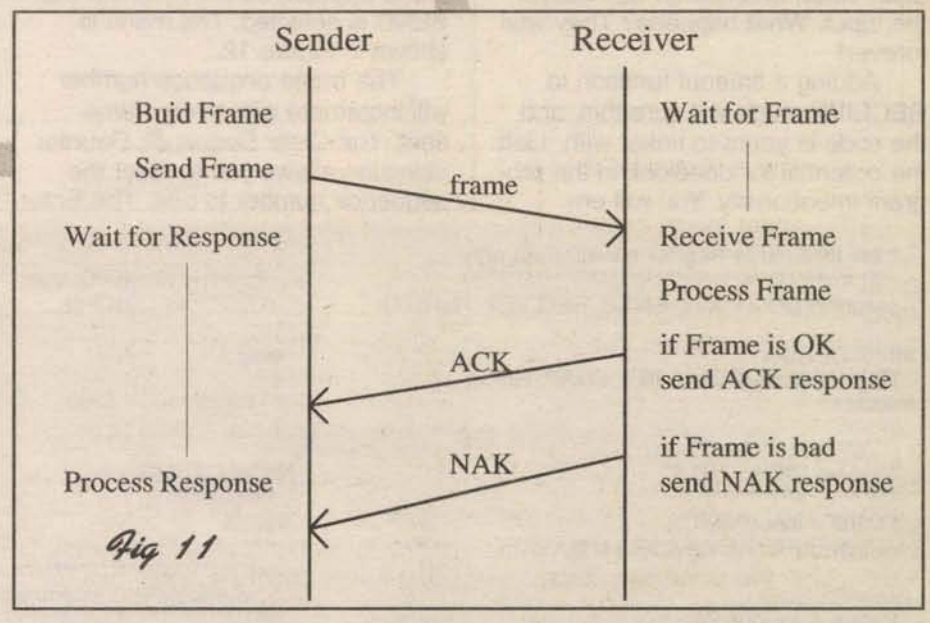
DL1.EXE is the executable file for our simple Stop and Wait protocol. The C source code is given in DL1.C and DL1.H. The program starts executing in Terminal Mode, and can transmit and receive characters at 19.2K baud on COM1. The best way to experiment with

```
/*****************************************************
* Filename    : dl1.c
* Description :    Data Link Layer serial communica-
* tions routines.
* Note: Compile with Large or Small Memory Model
*
* The functions in this module from datalink.c are:
*
* void interrupt DataLinkISR(void);
* void interrupt (*OldISR)(void);
* int InitCom1(void);
* int DisableCom1(void);
* int CheckBytes(void);
* int ReadByte(void);
* void WriteByte(unsigned char);
*
* New functions developed in modules are:
*
* void BuildFrame(char*);
* int GetFrame(void);
* void PutFrame(void);
* int CheckFrame(int);
* void FlushQueue(void);
* void SendAckResponse(void);
* void SendNakResponse(void);
*
* (c) 1997 Jeff Stefan
*
*****************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <conio.h>
#include <string.h>

#include "datalink.h"
#include "dl1.h"

#define ESCAPE 0x1b
#define SUCCESS 0
#define FAILURE -1

#define DEBUG
#define HEX_DISPLAY

/*********************
* Function prototypes
*********************/
void interrupt DataLinkISR(void);
void interrupt (*OldISR)(void);
int InitCom1(void);
int DisableCom1(void);
int CheckBytes(void);
int ReadByte(void);
void WriteByte(unsigned char);
void BuildFrame(char *);
int GetFrame(void);
void PutFrame(void);
int  CheckFrame(int);
void FlushQueue(void);
void SendAckResponse(void);
void SendNakResponse(void);


/************************
* Input queue variables
************************/
unsigned char InputQueue[MAX_QUEUE_SIZE];
unsigned char
*HeadPtr,*TailPtr,*StartOfQueue,*EndOfQueue;
unsigned char PicVal=0x00;

/*********************************
* Input Message Buffer and index
*********************************/
unsigned char InputMsgBuff[MAX_QUEUE_SIZE];
int BufIdx=0;

/***********************************
* Frame declarations
***********************************/
FRAME InFrame;
FRAME OutFrame;
unsigned char Buffer[MAX_BUFFER_SIZE];
char DataToSend[128];

typedef enum MODE
{TERMINAL,SEND,RECEIVE,EXIT};
int Mode = TERMINAL;

#ifdef DEBUG
  unsigned char Dbg;
  unsigned int  DbgCnt=0;
#endif

int main()
{
int InChar    = 0x00;
int Done      = 0;
int Input     = 0;
int Selection = 0;
int FrameLen  = 0;
int Status    = 0;
int i;

  clrscr();


/*****************************************************************
**
  * Init COM1 to 9600 baud, 8 data bits, 1 stop bit, and
no parity

*****************************************************************
**/
  InitCom1();

  /*****************************
  * Init the input queue pointers
  *****************************/
  HeadPtr    = TailPtr = StartOfQueue = InputQueue;
  EndOfQueue = StartOfQueue + MAX_QUEUE_SIZE;

  /*****************************
  * Main communications loop
  *****************************/
  while(!Done)
  {
    switch(Mode)
    {
      case TERMINAL:
        while(InChar != ESCAPE)
        {
          if(kbhit())
          {
            InChar = getch();
            if(InChar != ESCAPE)
            {
              WriteByte(InChar);
            }
          }
          else
          if(CheckBytes())
          {
            ReadByte();
          }
        }

        printf("\nEnter Mode: 0 = TERMINAL, 1 = SEND,
2 = RECEIVE 3 = EXIT\n");
        scanf("%d",&Input);
        switch(Input)
        {
          case TERMINAL:
            clrscr();
            InChar = NULL;
            break;

          case SEND:
#ifdef DEBUG
            printf("PacketData is: %s\n",DataToSend);
#endif
            clrscr();
            printf("—— Build and Send Packet ——\n");
```




Header
Data
0 1 2 3 4 5 6 7 8 9
3A 1 0 A 31 32 33 34 35 BB
Byte 0 : Start character
Byte 1: Sequence Number
Byte 2: ACK byte (zero for transmitter)
Byte 3: Frame Length
Byte 4 : ASCII 1
Byte 5: ASCII 2
Byte 6: ASCII 3
Byte 7: ASCII 4
Byte 8: ASCII 5
Byte 9: Checksum
Fig 10





the protocol is to run DL1.EXE on two machines connected with a three wire cable on COM1. (The cable diagrams were listed in Part 1 of this article.) When the Escape key is pressed, the following menu appears:

Enter Mode: 0 = TERMINAL, 1 = SEND, 2 = RECEIVE 3 = EXIT

Select SEND on one computer and RECEIVE on the other. Remember, DL1.EXE is a bare bones program and contains very little error checking. If you ask both machines to RECEIVE, you're stuck!

There are no timeouts added to RECEIVE mode. This is a classic example of an interprocess communications and multitasking operating

```
        printf("— 1. Clear Sequence Counter    —\n");
        printf("— 2. Enter Packet Data         —\n");
        printf("— 3. Display Packet            —\n");
        printf("— 4. Send Packet               —\n");
        printf("— Esc Exit Menu                —\n");
        printf("————————————————————————————\n");

        Selection = getch();
        switch(Selection)
        {
         /*************************
         * Clear sequence counter
         *************************/
         case '1':
          clrscr();
          OutFrame.Header.SeqNum = 0x00;
          printf("Sequence Number Cleared\n");
          printf("\nPress Esc...\n");
          break;

         /*********************
         * Enter packet data
         *********************/
         case '2':
          clrscr();
          printf("Enter Packet Data: \n");
          scanf("%s",DataToSend);
          BuildFrame(DataToSend);
          flushall();
          printf("\nPress Esc...\n");
          break;

         case '3': // display packet
          clrscr();
                printf("OutFrame.Header.StartChar =
[%X]\n",OutFrame.Header.StartChar);
          printf("OutFrame.Header.SeqNum    =
[%X]\n",OutFrame.Header.SeqNum);
          printf("OutFrame.Header.AckByte   =
[%X]\n",OutFrame.Header.AckByte);
          printf("OutFrame.Header.FrameLen  =
[%X]\n",OutFrame.Header.FrameLen);

          printf("OutFrame.Data.PktData =
%s\n",OutFrame.Data.PktData);
          printf("OutFrame.Data.ChkSum  =
[%X]\n",OutFrame.Data.ChkSum);
          break;

         case '4':  // send packet
          clrscr();
          PutFrame();

          /*********************
          *  get response frame
          *********************/
          while(!(CheckBytes()))
           ;
          FrameLen = GetFrame();
          Status = CheckFrame(FrameLen);
          break;

         default:
          break;
        }
        InChar = NULL;
        break;

       case RECEIVE:
        printf("Receiving.....\n");
        while(!(CheckBytes()))
          ;
        FrameLen = GetFrame();
        Status = CheckFrame(FrameLen);
        if(Status == SUCCESS)
        {
         SendAckResponse();
        }
        else
        {
         SendNakResponse();
        }

#ifdef DEBUG
        printf("\nExiting RECEIVE case\n");
#endif
        InChar = NULL;
        break;

       case EXIT:
        Done = 1;
        break;

       default:
        break;
       }
      break;

      default:
       break;
    }
   }
  DisableCom1();

#ifdef DEBUG
   printf("Debug Count = %d\n",DbgCnt);
#endif

  return(0);
  }

/*************************************************************
*: InitComPort : initializes COM port.
*
*************************************************************/
int InitCom1()
{
unsigned char PortData;

 /************************************
 * Set Line Control Register DLAB bit 7
 * to enable baudate initialization.
 ************************************/
 PortData = inportb(COM1_LINE_CTRL_REG);
 outportb(COM1_LINE_CTRL_REG, SET_DLAB |
PortData);

#ifdef DEBUG
 Dbg = inportb(COM1_LINE_CTRL_REG);
#endif

 /************************************
 * Write baudrate value to Reg0 and Reg1
 ************************************/
 outport(COM1,BAUD_192K);

 /************************************************
 * save the old interrupt service routine at INT4
 ************************************************/
 OldISR = getvect(IRQ_4);

 /************************************
 * use setvect() to point to com ISR
 ************************************/
 setvect(IRQ_4,DataLinkISR);

 /************************************
 * Setup Line Control Reg for eight data
 * bits, one stop bit, and no parity.
 ************************************/

outportb(COM1_LINE_CTRL_REG,EIGHT_DATA_BIT
S |
       ONE_STOP_BIT | NO_PARITY);

#ifdef DEBUG
  Dbg = inportb(COM1_LINE_CTRL_REG);
#endif

 /************************************
 * Set the Modem Control Register
 ************************************/
 outportb(COM1_MODEM_CTRL_REG,0x0b);

 /************************************************
```

```
---------- Build and Send Packet ----------
-- 1. Clear Sequence Counter
-- 2. Enter Packet Data
-- 3. Display Packet
-- 4. Send Packet
-- Esc Exit Menu
-------------------------------------------
```

Fig 12

system problem: deadlock.

A deadlock occurs when two processes are waiting for input from each other, and neither can deliver the input. What happens? They wait forever!

Adding a timeout function to RECEIVE mode will cure this, and the code is yours to tinker with. I left the potential for deadlock in the program intentionally. You will encounter deadlock conditions several times in your programming career.

The Build and Send Packet menu appears on the machine that SEND is selected. The menu is shown in Figure 12.

The frame sequence number will increment with every frame sent. The Clear Sequence Counter selection allows you to reset the sequence number to one. The Enter Packet Data selection allows you to enter data from the keyboard.

After the data is entered, the frame is automatically built and displayed. To examine the packet at any time, select Display Packet. To send the packet, select Send Packet.

After the packet is transmitted, the receiver displays the packet, then issues a NAK or ACK response packet. The response packet is displayed at the sender. After the receiver processes and responds to an incoming packet, it automatically returns to TERMINAL mode. Press Escape on the sending computer to return to the main menu.

You can use this simple protocol and the source code as a springboard for your own designs. You can expand the header structure to include command bytes and process ID bytes.

You can add a transport and application layer to transfer files to and from different machines, or you can implement your own remote procedure calls.

Working with communications protocols can be frustrating at times, but they're worth the effort to understand. Working with the simple protocol provided here will help you understand, analyze, and appreciate more sophisticated and complex protocols. **NV**

```
  * set Int Enable Reg for rcv interrupt only
  ********************************************/
  outportb(COM1_INT_ENAB_REG,SET_RxRDY);

#ifdef DEBUG
  Dbg = inportb(COM1_INT_ENAB_REG);
#endif

  /***********************
  * set up PIC for INT4
  ***********************/
  PicVal = inportb(PIC);
  outportb(PIC,PicVal & IRQ4_MASK);

#ifdef DEBUG
  Dbg = inportb(PIC);
#endif

  /*************************************
  * read char from port to flush input
  *************************************/
  inportb(0x3f8);

#ifdef DEBUG
  Dbg = inportb(0x3f8);
  Dbg = inportb(0x3f9);  // int enab
  Dbg = inportb(0x3fb);  // line ctrl
  Dbg = inportb(0x3fc);  // modem ctr
  Dbg = inportb(0x3fd);  // line status
  Dbg = inportb(0x3fe);  // modem status
  Dbg = inportb(0x21);   // PIC
#endif
  return(0);
}

/*************************************************************
*: DisableCom1 :  disables COM1 port.
*
*************************************************************/
int DisableCom1()
{
  /***************************************
  * restore old interrupt service routine
  ***************************************/
  setvect(IRQ_4,OldISR);

  /************************
  * restore old PIC value
  ************************/
  PicVal = inportb(PIC);
  outportb(PIC,PicVal | DISABLE_IRQ4);

#ifdef DEBUG
  Dbg = inportb(PIC);
#endif

  return(0);
}

/*************************************************************
*: DataLinkISR : serial port interrupt service routine.
*
*************************************************************/
void interrupt DataLinkISR(void)
{

unsigned char InChar;

 enable();

 InChar = inportb(COM1);
 if(TailPtr == EndOfQueue)
 {
   TailPtr = StartOfQueue;
   *TailPtr++ = InChar;
 }
 else
 {
   *TailPtr++ = InChar;
 }

#ifdef DEBUG
  DbgCnt+=1;
#endif

 /*****************************
 *  re-enable PIC interrupts
 *****************************/
 outportb(EOI,EOI);
}

/*************************************************************
*: CheckBytes : checks if queue contains unread chars. * Returns zero in no bytes in queue, else returns
* number  of chars in queue.
*************************************************************/
int CheckBytes(void)
{
  /*****************************************
  * check if head pointer == tail pointer
  * if ==, no chars are in buffer: return(0);
  * num chars = abs diff from head to tail
  * return num chars.
  *****************************************/
  if(HeadPtr - TailPtr)
  {
    return(abs(HeadPtr - TailPtr));
  }
  else
  {
    return(0);
  }
}

/*************************************************************
*: FlushQueue: Reassigns input queue head and tail pointers.
*
*************************************************************/
void FlushQueue()
{
  HeadPtr    = TailPtr = StartOfQueue = InputQueue;
  EndOfQueue = StartOfQueue + MAX_QUEUE_SIZE;
}

/*************************************************************
*: ReadByte : reads a byte from the queue
*
*************************************************************/
int ReadByte(void)
{
int ByteCount=0;
  /*****************************************
  * check if head pointer == tail pointer
  * if ==, no chars are in buffer: return(0);
  *****************************************/
  if(HeadPtr >= TailPtr)
  {
    HeadPtr = StartOfQueue;
    return(0);
  }
  else
  {
#ifndef HEX_DISPLAY
    printf("%c",*HeadPtr);
#endif
#ifdef HEX_DISPLAY
    printf("[%X]",*HeadPtr);
#endif

    HeadPtr++;
  }
  return(0);
}

/*************************************************************
* WriteByte : writes a byte out the serial port
*
*************************************************************/
void WriteByte(unsigned char CharToSend)
{
 /*************************************
 * Wait for the transmit flag to clear
 *************************************/
 while((inportb(COM1_LINE_STAT_REG) & 0x20) ==
0)
 {
   ;
 }
 /*****************
 * Output the byte
 *****************/
 outportb (COM1,CharToSend);
}

/*************************************************************
*: BuildFrame: builds a frame to send from a buffer.
*
*************************************************************/
void BuildFrame(char DataToSend[])
{
int PktLen = 0x00;
int ChkSum = 0x00;
int i;
char *ptr;

  /*************************************
  * increment packet sequence number
  *************************************/
  OutFrame.Header.SeqNum+=1;

  /************************
  * set ACK byte to zero
  ************************/
  OutFrame.Header.AckByte = 0x00;

  /*********************
  * load packet data
  *********************/
  strcpy(OutFrame.Data.PktData,DataToSend);

  /*****************************
  * calculate packet length
  *****************************/
  ptr = OutFrame.Data.PktData;
  while(*ptr++)
  {
    PktLen+=1;
  }
  /**********************
  * add checksum byte
  **********************/
  PktLen+=1;

  /**********************
```

```
  * add header length
  *********************/
  PktLen+=HEADER_SIZE;

  /*********************
  * load length byte
  *********************/
  OutFrame.Header.FrameLen = (unsigned
char)PktLen;

  /************************
  * load start character
  ************************/
  OutFrame.Header.StartChar = ':';

  /************************
  * calculate checksum
  ************************/
  ChkSum = OutFrame.Header.StartChar;
  ChkSum+= OutFrame.Header.SeqNum;
  ChkSum+= OutFrame.Header.AckByte;
  ChkSum+= OutFrame.Header.FrameLen;

  /*****************************
  * add packet data checksum
  *****************************/
  for(i=0;i<PktLen - HEADER_SIZE;i++)
  {
    ChkSum+= OutFrame.Data.PktData[i];
  }
  ChkSum = ~(ChkSum);
  OutFrame.Data.ChkSum = ChkSum;

#ifdef DEBUG
  printf("\nOutFrame.Header.StartChar =
[%X]\n",OutFrame.Header.StartChar);
  printf("OutFrame.Header.SeqNum    =
[%X]\n",OutFrame.Header.SeqNum);
  printf("OutFrame.Header.FrameLen  =
[%X]\n",OutFrame.Header.FrameLen);
  printf("OutFrame.Header.AckByte   =
[%X]\n",OutFrame.Header.AckByte);

  printf("OutFrame.Data.PktData =
%s\n",OutFrame.Data.PktData);
  printf("OutFrame.Data.ChkSum  =
[%X]\n",OutFrame.Data.ChkSum);
#endif
}

/*************************************************************
*: GetFrame: Builds a frame from an incoming byte
stream.
*
*************************************************************/
int GetFrame()
{
unsigned char *ptr;
int i = 0x00;
int DataLen=0;
int Ctr;

  while(CheckBytes())
  {
    ReadByte();
  }
  /***************************************
  * transfer input queue to InputMsgBuff[]
  ***************************************/
  ptr = InputQueue;

  /************************
  * check for start char
  ************************/
  if(*ptr == ':')
  {
    /*******************
    * load start char
    *******************/
    InputMsgBuff[i++] = *ptr++;

    /************************
    * load sequence number
    ************************/
    InputMsgBuff[i++] = *ptr++;

    /*****************
    * load ACK byte
    *****************/
    InputMsgBuff[i++] = *ptr++;

    /*********************
    * load frame length
    *********************/
    InputMsgBuff[i++] = *ptr;
    DataLen = (int)*ptr;
    ptr++;

    /*********************
    * load data segment
    *********************/
    for(Ctr=(HEADER_SIZE+1);Ctr<DataLen;Ctr++)
    {
      InputMsgBuff[i++] = *ptr++;
    }
    /*******************
    * load checksum
    *******************/
    InputMsgBuff[i++] = *ptr;
  }
  else
  {
   DataLen = -1;
#ifdef DEBUG
    printf("Bad Packet Received!!\n");
#endif
    /*********************
    * Flush InputQueue
    *********************/
    FlushQueue();
  }
  return(DataLen);
}

/*************************************************************
*: PutFrame: Calls WriteByte() to output a frame to a
serial
*          port.
*************************************************************/
void PutFrame()
{
int i;
  /*******************
  * write header
  *******************/
  WriteByte(OutFrame.Header.StartChar);
  WriteByte(OutFrame.Header.SeqNum);
  WriteByte(OutFrame.Header.AckByte);
  WriteByte(OutFrame.Header.FrameLen);

  /*********************
  * write data
  *********************/
  for(i=0;i<OutFrame.Header.FrameLen -
(HEADER_SIZE+1);i++)
  {
    WriteByte(OutFrame.Data.PktData[i]);
  }
  WriteByte(OutFrame.Data.ChkSum);
}

/*************************************************************
*: CheckFrame: Checks a frame's checksum byte.
*
*************************************************************/
int CheckFrame(int FrameLen)
{
unsigned char ChkSum=0;
unsigned char PktChkSum=0;
int Result=0;
int i;
char *ptr;

  ptr = InputMsgBuff;
  for(i=0;i<FrameLen-1;i++)
  {
    ChkSum += *ptr++;
  }
  PktChkSum = *ptr;
  Result = ChkSum ^ PktChkSum;
  if(Result != 0xff)
  {
    Result = FAILURE;
  }
  else
  {
    Result = SUCCESS;
    InFrame.Header.SeqNum = InputMsgBuff[1];
  }
  return(Result);
}

/*************************************************************
*: SendAckResponse: Builds and sends an ACK frame.
*
*************************************************************/
void SendAckResponse()
{
int PktLen = 0x00;
int ChkSum = 0x00;
int i;
char *ptr;

  OutFrame.Header.SeqNum =
InFrame.Header.SeqNum;
  OutFrame.Header.AckByte = ACK;
  PktLen+=1;
  PktLen+=HEADER_SIZE;
  OutFrame.Header.FrameLen = (unsigned
char)PktLen;
  OutFrame.Header.StartChar = ':';

  ChkSum = OutFrame.Header.StartChar;
  ChkSum+= OutFrame.Header.SeqNum;
  ChkSum+= OutFrame.Header.AckByte;
  ChkSum+= OutFrame.Header.FrameLen;
  ChkSum = ~(ChkSum);
  OutFrame.Data.ChkSum = ChkSum;

#ifdef DEBUG
  printf("\nOutFrame.Header.StartChar =
[%X]\n",OutFrame.Header.StartChar);
  printf("OutFrame.Header.SeqNum    =
[%X]\n",OutFrame.Header.SeqNum);
  printf("OutFrame.Header.FrameLen  =
[%X]\n",OutFrame.Header.FrameLen);
  printf("OutFrame.Header.AckByte   =
[%X]\n",OutFrame.Header.AckByte);
  printf("OutFrame.Data.ChkSum  =
[%X]\n",OutFrame.Data.ChkSum);
#endif

  WriteByte(OutFrame.Header.StartChar);
  WriteByte(OutFrame.Header.SeqNum);
  WriteByte(OutFrame.Header.AckByte);
  WriteByte(OutFrame.Header.FrameLen);
  WriteByte(OutFrame.Data.ChkSum);

#ifdef DEBUG
  printf("\nACK Response Sent\n");
#endif
}

/*************************************************************
*: SendNakResponse: Builds and sends a NAK frame.
*
*************************************************************/
void SendNakResponse()
{
int PktLen = 0x00;
int ChkSum = 0x00;
int i;
char *ptr;

  OutFrame.Header.SeqNum =
InFrame.Header.SeqNum;
  OutFrame.Header.AckByte = NAK;
  PktLen+=1;
  PktLen+=HEADER_SIZE;
  OutFrame.Header.FrameLen = (unsigned
char)PktLen;
  OutFrame.Header.StartChar = ':';

  ChkSum = OutFrame.Header.StartChar;
  ChkSum+= OutFrame.Header.SeqNum;
  ChkSum+= OutFrame.Header.AckByte;
  ChkSum+= OutFrame.Header.FrameLen;
  ChkSum = ~(ChkSum);
  OutFrame.Data.ChkSum = ChkSum;

#ifdef DEBUG
  printf("\nOutFrame.Header.StartChar =
[%X]\n",OutFrame.Header.StartChar);
  printf("OutFrame.Header.SeqNum    =
[%X]\n",OutFrame.Header.SeqNum);
  printf("OutFrame.Header.FrameLen  =
[%X]\n",OutFrame.Header.FrameLen);
  printf("OutFrame.Header.AckByte   =
[%X]\n",OutFrame.Header.AckByte);
  printf("OutFrame.Data.ChkSum  =
[%X]\n",OutFrame.Data.ChkSum);
#endif

  WriteByte(OutFrame.Header.StartChar);
  WriteByte(OutFrame.Header.SeqNum);
  WriteByte(OutFrame.Header.AckByte);
  WriteByte(OutFrame.Header.FrameLen);
  WriteByte(OutFrame.Data.ChkSum);

#ifdef DEBUG
  printf("\nNAK Response Sent\n");
#endif
}
```